1

An Event-Driven Model-View-Controller Framework for Smalltalk

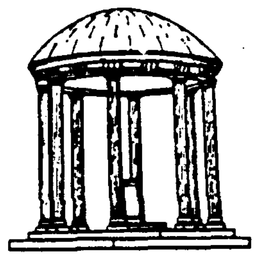
TR89-025

June, 1989

N00014-86-K-0680

Yen-Ping Shan

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175



A TextLab Report

Portions of this research were supported by the National Science Foundation, Grant #IRI-85-19517 and the Army Research Institute, Contract #MDA903-86-C-0345. UNC is an Equal Opportunity/Affirmative Action Institution.

An Event-Driven Model-View-Controller Framework for Smalltalk

Yen-Ping Shan

June 19, 1989

Abstract

The Smalltalk Model-View-Controller (MVC) user interface paradigm uses polling for its input control. The polling loops consume CPU cycles even when the user is not interacting with the interface. Applications using Smalltalk as their front-end often suffer unnecessary performance loss. This paper presents a prototype event-driven MVC framework to solve these problems. A solution to the compatibility problem is also provided to allow interface objects built under both polling and event-driven mechanisms to be used by each other with no modification and no performance penalty.

1 Introduction

The Model-View-Controller paradigm [Adams 88, Krasner 88] provides the framework for most Smalltalk-80 [Goldberg 83] user interfaces. Within this framework, the user interface objects interact with the end user by polling the states of the input devices and responding to the state changes.

Polling is a source of unnecessary performance loss. The polling loops must always be active in order not to miss any action performed by the user. When one is developing systems with multiple processes, this becomes a serious problem. For example, an application with a polling user interface may fork an agent process to handle the transactions to a remote database and to manage the local cache. Since the user interface process must keep polling even when the user is not interacting with the system (for example, the user is waiting for a transaction to finish), it consumes the CPU cycles that should have been spent on the database agent process. Moreover, the existence of the database agent process makes the interface less responsive. The situation is aggravated when the database is running on the same machine as the user interface.

This deterioration of performance can be avoided if the user interface is built on top of an event-driven mechanism that does not poll[1]. However, one must be cautious in making such a fundamental change. While switching to an event-driven mechanism is beneficial, throwing away all of the existing user interfaces and rebuilding them under a new mechanism for better performance is too high a cost. Reusability is among the most important features of object-oriented programming. If the new event-driven mechanism does not allow us to reuse the work done with the polling mechanism, it would be much less useful.

This paper presents a prototype event-driven interface framework that not only solves the performance problem but also allows:

- interfaces built with the polling mechanism to co-exist with ones that are built with the event-driven mechanism. (For example, an event-driven directory browser could co-exist with the standard Smalltalk system browsers.)
- interface objects built with both mechanisms to be reused by each other. (For example, within a polling environment one could use an event-driven spread-sheet which in turn uses a polling menu.)

Additionally, no modification to existing code is required and there is no performance penalty.

The next section gives a brief overview of both the polling and event-driven mechanisms. In section 3, further motivation for having an event-driven mechanism is provided. Section 4 describes the design and implementation of the prototype event-driven MVC framework. Section 5 discusses the solution to the compatibility problem. The last section gives the status of the implementation.

[1]An alternative is to implement a Time-Sharing Citizenry [Schiffman 88] mechanism within the Smalltalk itself.

2 Background

Polling

A system that supports the polling mechanism often maintains a globally accessible table of the states of the devices. In Smalltalk, this table is an instance of *InputSensor* and is accessible through a global variable called *Sensor*. A typical interface object will have loops that poll the relevant table entries. When a state change is sensed, the case statement in the loop invokes a routine to process the change. This routine can change the state of the underlying application, give feedback to the user, or transfer control to another loop to detect further state changes. For example, a Smalltalk *PopUpMenu* is often invoked by a loop that senses mouse button presses. Control is then passed to the *PopUpMenu* polling loop which tracks the cursor position and highlights the proper portion of the menu when the user drags the cursor.

The control structure of a polling interface is characterized by a tree of loops. Each loop in the tree keeps control while certain conditions are satisfied (for instance, the cursor stays within a rectangle area), and polls the children loops to see whether they want control. A child loop that wants control can grab it, and later return control to its parent loop when its looping condition is no longer satisfied.

Event-Driven

An event-driven mechanism [Newman 79] usually consists of three major components: a set of event generators, an event queue that buffers the events in sequence, and an event dispatching mechanism that removes the events one at a time from the queue and sends them to the corresponding event handlers. An event has a name or number that identifies the nature of the interaction plus several data values that characterize the interaction.

A typical event-driven interface has a single event-fetching loop. The execution of the loop is suspended when the event-fetching statement in the loop tries to fetch from an empty event queue and resumes when new events arrive.

An event-driven interface program registers a number of event handlers with the event dispatching mechanism. For each handler, a list of interested event types is specified. When an interesting event happens, the dispatching mechanism activates the corresponding handler to process it.

3 Why Event-Driven?

Besides the benefits in performance mentioned above, the event-driven mechanism provides a better trace of input devices. With the polling mechanism, when a system is heavily loaded, it can miss a state change (for example, a button click) because the polling loop is not at the condition statement when the change happened. This does not happen with event-driven model where all the events are buffered. An application has the freedom to discard events when it cannot process them as fast as they come (this is seldom the case, though); it can also control when the events should be discarded and which one to discard. This is in contrast to the polling mechanism where state changes are ignored, depending on the system load and the execution timing of the statements in the polling loop.

The event-driven mechanism also makes possible implementation of certain applications that could not be done within a polling paradigm. For instance, with the prototype event-driven mechanism described in the next section, the author was able to develop a package that allows users running Smalltalk on different machines to share visual workspaces. The package is general in that a user can select any event-driven application and then share both control and the visual display with other users.

4 A Prototype Event-Driven MVC

This section describes the three major components–the event generator, the event queue, and the event dispatching mechanism–for a prototype event-driven framework which preserves the structure and the semantics of the MVC paradigm.

4.1 Event Generator

An event generator is responsible for generating events and placing them on the event queue. Beneath the Smalltalk virtual machine, the input devices are handled by an event-driven (more precisely interrupt-driven) mechanism; consequently, the problem of creating an event generator is reduced to identifying the place where Smalltalk changes its state table and inserting code to generate the events. Smalltalk acquires the primitive input events from the virtual machine through the method "primitiveInputWord" and updates its state table in the *InputState* class. The code inserted in the "run" method of the *InputState* class interprets the primitive input events to construct the events used by the framework. Methods are also added to the *InputState* to control event generation.

4.2 Event Queue

The implementation of the event queue is straightforward. The Smalltalk *SharedQueue* provides most of the functionality needed by the event queue, including suspending processes that try to fetch from an empty queue. The *EventQueue*, a subclass of *SharedQueue*, implements methods to control the queue and to handle queue overflow.

4.3 Event Dispatching and the MVC

The event dispatching mechanism is more subtle and the decisions made here affect compatibility. The goal is not just to produce a mechanism that delivers the events to the right event handlers, but also to ensure that the created event-driven interfaces are compatible with polling interfaces.

The "superView-subView" relation in the Smalltalk *View* class provides the base for event dispatching. A *View* in a structured picture can contain other *Views* as sub-components. These sub-components are called "subViews." A *View* can be a subView of only one *View*–its "superView." The set of *Views* in a structured picture forms a hierarchy. In the prototype, all screen objects inherit from a subclass of *View* called *Mode*. When a *Mode* receives an event, it checks to make sure the event is intended for it (usually by comparing the coordinates of the event with its display box) and asks all of its "subModes," starting from the topmost one, to process the event. (The "subModes" are stored in the instance variable "subViews" inherited from *View*.) If none of the subModes are interested in the event, it then tries to process the event itself. If it is not interested in the event, it returns the event as un-processed to its "superMode" (stored in the instance variable "superView," also inherited from *View*). A *Mode* delegates responsibility for processing events to its event handler, which is stored in the instance variable "controller," defined by the MVC paradigm. In the prototype, the *Mode* defines a number of new methods to provide better clipping and windowing behavior.

The one *Mode* in the hierarchy that has no superMode is called the "rootMode." It is an instance of *RootMode* class where the event-fetching loop is defined. A typical application would have a single *RootMode* and a hierarchy of *Modes*. To allow multiple active applications, a built-in mechanism is provided in *RootMode* to guarantee that no two *RootModes* will attempt to access the event queue at the same time.

The above arrangement creates an event-driven framework which preserves the structure of the MVC paradigm. It allows the Smalltalk "MVC inspector" to be used without any modification. The event-driven framework also preserves the semantics of the MVC paradigm. The *View* is still responsible for visual aspects of the structured picture, and the *Controller* (now an event handler) is still in charge of the user interaction. Since both the structure and semantics of the MVC paradigm are preserved by the event-driven framework, we term it "event-driven MVC."

5 Compatibility

The problem of compatibility comes from having two active mechanisms (event-driven and polling) present at the same time. This can be viewed as a control switching problem. At any time, one would like to make sure that the mechanism in control corresponds to the type of object that the user is interacting with, and that there is no interference from the other mechanism. Knowing *when* and *how* to switch between the two mechanisms is the key to achieving compatibility.

5.1 Definition of the Problem

Strings of capital letters are used to present the problems concisely. The string XY denotes that an object built with mechanism Y is running in an environment built with mechanism X. Each letter can either be P, denoting the polling mechanism, or E, denoting the event-driven mechanism. For example, the string PE represents the situation of an event-driven object running under an environment that is controlled by a polling object. The string PEP would describe a polling interface object running under an event-driven environment which in turn is running under another polling environment. The spread-sheet example used in the Introduction section is modeled by this string. A string of PPEPEEPE represents a highly nested interface with event-driven and polling objects inter-mixed.

Although the compatibility problem may look complicated at the first glance, it is regular. Notice that if the sub-problems PP, EE, PE, and EP can be solved, all of the more complicated problems are merely concatenations of these four basic cases. Since the first two sub-problems are trivial, only the last two need further consideration.

5.2 When to Switch

For reasons of performance and preventing interference, one must avoid having two mechanisms running at the same time whenever possible. This precludes the use of a single mechanism as the master mechanism which determines when to switch to a slave mechanism. The only choice left is to have the X, the environment mechanism, in each XY pair, determine the switches.

5.3 Sandwiching

A technique, called "Sandwiching," which inserts an invisible layer between a pair XY is used to provide solutions to both the EP and PE cases. After the invisible layer (named "ham") is included in the representation, the structure becomes XHY. Figure 1 shows an EHP sandwich. The purpose of the "ham" is to make X feel like Y is built with the same mechanism as it is and vice versa. If the "ham" is well designed, no modification to either X or Y is necessary in order to have them running together. Therefore, the problem of how to switch is addressed by the design of the "ham."

5.4 How to Switch: Case EHP

The "ham" for this case is a *Mode* with a special event handler (controller) which suspends event generation and flushes the event queue when certain conditions (for example, an "EnterWindow" event is received) indicate that

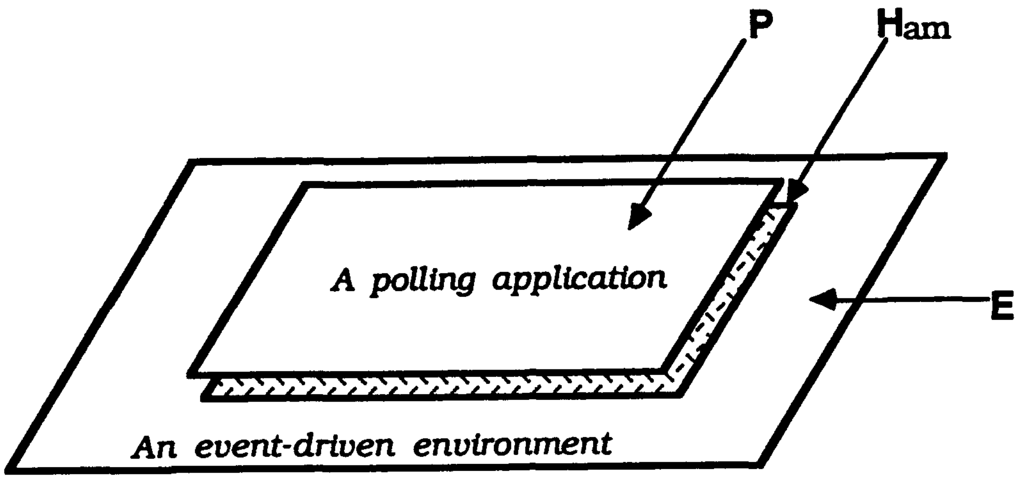


Figure 1: An EHP sandwich.

the polling application P should be in action. The "ham" then brings itself, and therefore the P, to the front of the display (so that nobody obscures them) and passes the control to the controller of the top view of P. When control is returned, it resumes event generation.

The choice of making *Mode* a subclass of *View* shows another benefit besides reusing code. It makes the "ham" easy to use. Since the "ham" inherits the behavior of *View*, P can treat it as an ordinary polling *View*, and E can treat it as an event-driven *Mode*. To construct the sandwich, one simply creates a "ham," attaches to it the polling application as its only sub-View, and attaches the "ham" to the underlying event-driven environment. No modification of either P or E is required.

5.5 How to Switch: Case PHE

There are two types of E, self-contained event-driven applications with their own event-fetching loops (with *RootModes*) and those that are without an event-fetching loop. For both types, the "ham" must provide the event-fetching loop. It may not be obvious why an event-fetching loop is needed for the self-contained applications that already have one. The reason comes from an important distinction between event-driven and polling applications. A polling application returns control to its parent when the condition for looping is not satisfied, but an event-driven application does not. The only time an event-driven application breaks its event-fetching loop and returns is when it terminates. A simple-minded "ham" that activates the event generation, passes control to the event-fetching loop of the event-driven application, and waits for it to return will not work because there is no guarantee that the control will come back.

Certainly, one can modify the event-driven application so that it returns control under certain condition (for example, "LeaveWindow" event received), but this breaks the promise of no modification. Another alternative is to let the "ham" and the application run as two processes and have the "ham" suspend and resume the application process. This is also not satisfactory because it introduces both the complexity of inter-process communication and the performance loss due to the looping nature of the "ham" process.

A technique called "loop merging" is employed. The event-fetching loop in the application is merged with the polling loop in the "ham," as shown in Figure 2. This is done by copying the code in the event-fetching loop and inserting it into the "ham" polling loop. The merged loop, then, serves as the event-fetching loop. The real event-fetching loop of the application is never executed. The merged loop in the "ham" checks the device state changes interesting to the "ham" (for example, see if cursor is still in), fetches an event from the event queue, and asks the application to process the event (by sending the event to the "topMode" of E). The "ham" enables the event generation before it enters the merged loop, and disables the event generation after it leaves the loop.

The merged loop is suspended when there is no event in the queue. This improves the performance of other processes since no CPU cycles are wasted in the useless polling in the "ham." The merged loop also transfers control properly. When the user switches to another application (often by moving the cursor onto that application), there are always events generated by the user's action to wake up the merged loop for it to return the control to its parent (the P). The parent can ,then, assign control to the newly selected application.

One can also insert code into the merged loop to ensure the event-driven application conforms to the windowing behavior of the underlying polling environment. For example, the Smalltalk interface (a P) uses the blue button (the right mouse button) for windowing control (e.g., resize, move, collapse). The inserted statements in the merged loop, as shown in Figure 2, can check the status of the blue button and activate the "ScheduledBlueButtonMenu" when the button is pressed. The user can, then, manipulate the window of

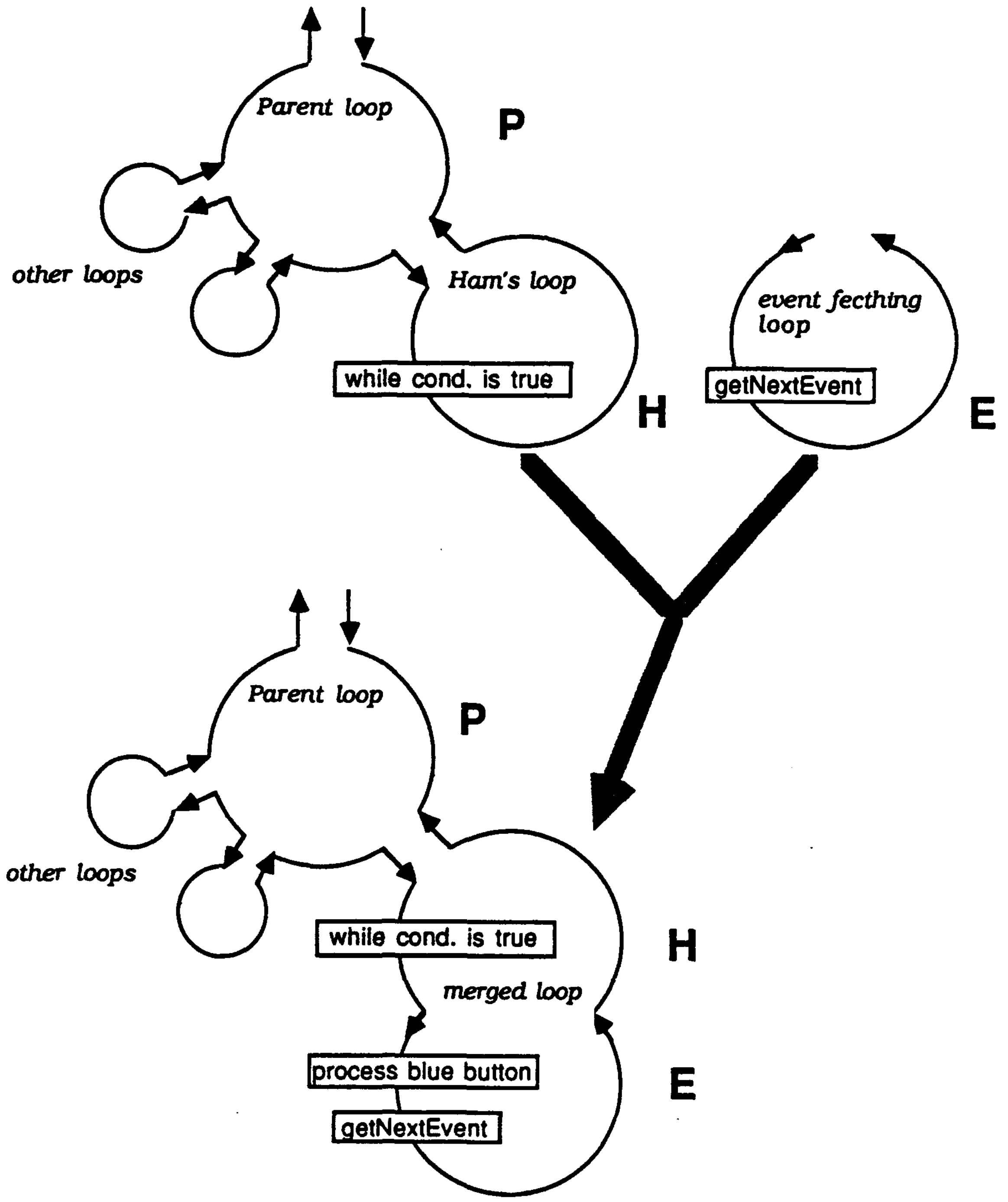


Figure 2: Loop merging

the event-driven application just as it were a Smalltalk *StandardSystem View*.

6 Conclusion

The event-driven MVC framework described above preserves both the structure and the semantics of the MVC paradigm. It not only allows efficient user interfaces to be built, but also provides necessary compatibility with the polling interfaces.

A prototype of the event-driven MVC framework has been built. Test interfaces built with it show better background process performance and cleaner program structure. Although no formal measurment has been done, the test interfaces can conserve over 30% of the CPU time for the background processes under the worst case (when the user is dragging a *Mode* clipped against the *Modes* surrounding it). All of them are as responsive, if not more so, than those built with the polling mechanism. Some of the test interfaces (for instance, the general shared visual workspace) cannot be built with the traditional polling mechanism of Smalltalk. The "Sandwiching" technique has been successfully applied to create interfaces that mix the Smalltalk user interface objects (text editor, debugger, menu, binary choice, etc.) with the event-driven interface objects. The author is currently using this prototype to develop a user interface management system for Smalltalk that supports direct manipulation user interfaces.

7 Acknowledgement

A number of organizations and people have contributed to the work reported here. The author is grateful to the National Science Foundation (Grant # IRI-85-19517) and the Army Research Institute (Contract #MDA903-86-C-0345) for their support of this research. John B. Smith, Rick Snodgrass, Matt Barkley, and Gordon Ferguson provided valuable comments and suggestions for this paper. The Textlab Research Group within the Department of Computer Science at the University of North Carolina at Chapel Hill has provided a provocative and supporting intellectual environment for this work. Finally, special thanks to Jonathan Eunice who motivated this research. Without his vision, this work would not have been done.

References

[Adams 38] Adams, S. S. *MetaMethods: The MVC Paradigm.* HOOPLA! Vol. 1, No. 4, July 1988.

[Goldberg 83] Goldberg, A. & Robson, D. *Smalltalk-80: the Language and Its Implementation.* Addison-Wesley, 1983.

[Krasner 88] Krasner, G. E. & Pops, S. T. *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80.* Journal of Object-Oriented Programming, Vol. 1, No. 3, August/September 1988. pp. 26-49.

[Newman 79] Newman W. M., & Sproull, R. F. *Principles of Interactive Computer Graphics.* McGraw-Hill, Inc., 1979.

[Schiffman 88] Allan M. Schiffman *Time-Sharing Citizenry for Smalltalk-80 under UNIX.* ParcPlace Newsletter, Vol. 1, No. 2, ParcPlace Systems, 1988. pp. 9-10.